Code Generation Strategies for Partitioned Systems

Julien DELANGE, Laurent PAUTET
TELECOM ParisTech - LTCI UMR 5141
46, rue Barrault
F-75634 Paris CEDEX 13, France
delange@enst.fr, pautet@enst.fr

Fabrice KORDON
LIP6, Univ. P & M. Curie
4 place Jussieu
75252 Paris Cedex 05, France
fabrice.kordon@lip6.fr

Abstract

Design and verification of Distributed Real-time Embedded (DRE) systems are crucial because a failure or a security problem may cause loss of life or the premature end of a mission. Code for such systems must be verified to avoid failures and errors. To improve dependability and reliability, the concept of partitioned architecture has been proposed. Partitioned architectures isolate partitions and reduce failures propagation. However, system's configuration as well as application code remain hand-written, which is error-prone, difficult to check and certify.

In this paper, we present an approach to automatically generate and configure partitioned systems. We model partitioned systems using a language suitable for real-time embedded systems. Then, we automatically generate code from models. Generated code follows the semantics of the model so it enforces specified requirements.

1 Introduction

Dependability and reliability are crucial topics for distributed, real-time and embedded (DRE) systems, especially when failures are mission or life-critical. These systems must be certified and verified.

Confidentiality, security and safety are also major topics: safety and security policies must be enforced across the system (from low to higher layers). We must ensure that entities can read or write only some data (security enforcement), perform only allowed operations and that failures are not propagated across the system.

To address these issues, partitioned architectures [1, 9] were introduced. They enforce isolation in term of space and time between software components. Partitioned systems improve dependability and reliability. However, verification of their requirements is made at run-time. Moreover, most of code remains hand-written, which is prone of error and makes system's verification difficult.

This papier introduces a new process development (illustrated in figure 1) that enables verification of system's security and safety requirements at design-time. At first, system's designer writes specifications and checks his/her requirements enforcement (step 1). Then, configuration as well as application code is automatically generated from the specifications (step 2). Finally, underlying executive, generated code and application code (the functional part of the system) are compiled together. The result is a functional system that enforces specifications requirements.

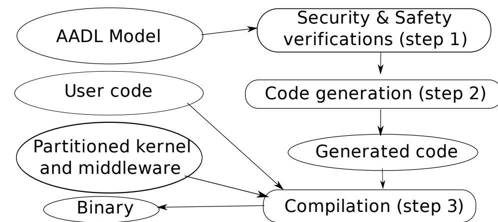

Figure 1. Development process using model and code generation

Such a design process must rely on a modeling language to describe system's architecture and properties. We selected the Architecture Analysis and Design Language (AADL), which is suitable to design DRE systems. Modeling and verification of safe and secure systems using AADL (step 1) has been already explored [4, 8]. Consequently, this work focuses on code generation from these models (step 2).

Our code-generation process is automatic: kernel's configuration code is created according to the model as well as application and configuration code for each partition. It automatically sets up kernel's and partition's services, configures resources (priorities of tasks, scheduling policy, etc.), which results in a more reliable and robust system.

2 Background

This section presents the AADL, the main principles of partitioned architectures and related work.

2.1 AADL

AADL is a component-centric language to model both software and hardware components. It focus on the definition clear block interfaces, and separates implementations from these interfaces.

An AADL description is made of *components*: software components (`data`, `thread`, `thread group`, `subprogram`, `process`), execution platform components (`memory`, `bus`, `processor`, `device`, `virtual processor`, `virtual bus`) and hybrid components (`system`). The `system` component is the topmost component in the model and aggregates other subcomponents (`process`, `processor`, ...).

The interface of a component is called its *type*. It provides *features* (e.g. communication ports). Components communicate by *connecting* their *features*.

The AADL defines the notion of *properties* that can be added to most elements (components, connections, features etc.). Properties are name/value pairs that specify constraints or characteristics to be applied to the system's components (execution time of a thread, bandwidth of a bus, etc.) An introduction to the AADL can be found in [7].

Compared to other modeling languages (like UML and its extensions[3]), AADL provides two major benefits for building DRE systems. First, it defines low-level abstractions including hardware descriptions. Second, hybrid system components help to refine the architecture as they can be detailed later in the design process.

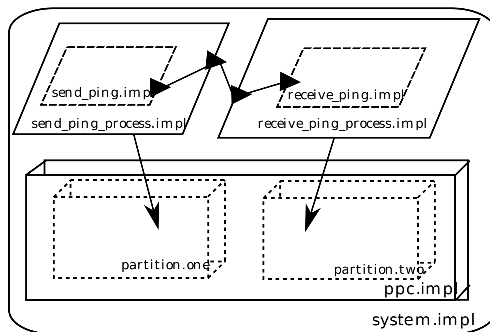

Figure 2. Partitioned producer/consumer system AADL

Partitioned architectures are modeled using `virtual processors` and `process` components. Both model a specific runtime (mainly a scheduler supporting time isolation) and a single address space (which corresponds to the space isolation). Security layers are defined using the `virtual bus` components. Security levels are declared for each component and connection and can be used to check the system against a security policy.

An example of AADL model is shown in figure 2. It represents a simple system which has two partitions that exchanging data (one partition sends *ping* to the other). Each process is bounded to a virtual processor. Virtual processors are part of the main processor (the physical one). We enrich the model with properties to describe system's behavior, defining safety and security requirements (scheduling, isolation enforcement, etc.).

2.2 Partitioned kernels and middleware

In the last years, new kernels and middleware architectures focusing on safety and security have been designed. Isolating software components in term of space and time is the main main concept elaborated for these architectures [1, 9]. Pieces of software are isolated in an entity called *partition* to avoid error or failure propagation among partitions. Each partition has its own address space to store its code or data, and a time budget (called *timeslice*) to execute its threads. A partition can neither access to the other partitions' data nor consume more time than allocated. In a partitioned system, isolation is achieved so each partition is executed as if it was on a single computer.

Partitioned kernels are small to ease their verification. Few services are provided: memory management (space isolation), scheduling of partition (time isolation), and some device drivers (such as clock driver for scheduling). These services, as well as the resources contained in each partition are created and allocated at initialization-time to avoid error at run-time. In addition, device drivers are not included in the kernel [6] to keep it as small as possible.

Partitioned middleware sends and receive data between partitions. It enforces data separation among the partitions and ensures isolation of communications.

2.3 Related work on code-generation

Many work has already been made on code generation, using several modeling languages[5]. On our side, we already designed a code-generator that outputs Ada or C code from AADL models [2]. This approach had many advantages regarding the needs of embedded real-time systems (reduction of dead-code, low memory footprint, etc.). Unfortunately, the underlying code-generation patterns neither cared about safety and security issues nor relied on traditional kernels which provide no safety and security mechanisms. We base our current work on this approach and design new code-generation patterns that meet these new requirements.

3 Code-generation strategies

In this section, we describe the main guidelines of our code generation patterns.

3.1 Partitioned architecture guidelines

Kernel and middleware services are configured according to the AADL model to enforce safety and security requirements. The code generator creates code to configure kernel's services for each partition. Moreover, it configures each partition and generates its code. Our code generation process is involved in kernel and partitions layers.

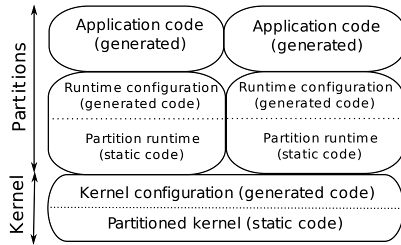

Figure 3. Architecture of generated code

Figure 3 shows the architecture of the application produced by our code generator. In the kernel layer, data structures are defined to configure kernel and middleware services (scheduler, communication ports, etc.). In this layer, resources are allocated for each partition and services are configured to enforce security and safety requirements. For example, we configure the inter-partition ports and generate data that describe allowed operations on each port.

We configure each layer and select required services only. We also create and allocate resources at initialization-time to fit with the requirements of partitioned architectures.

In the partition layer, we generate the application code for each partition. This code is loaded in each partition's address space and contains required services and data structures (tasks, data, mutexes and so on). Generated code contains all resources needed by the partition and initializes them and set appropriate properties (priorities, locking protocols, etc.) at initialization-time.

All generated data and entities are deduced from the AADL specification. The generation strategy avoids overhead in the code and reduces both dead code and memory footprint as much as possible. AADL models declare required resources and data so required allocated resources can be determined. This property is of particular interest in the embedded domain, where memory footprint is a major concern.

3.2 Code-generation patterns

An AADL model describes a system's architecture including its properties and requirements. The topmost component (`system`) represents the root of the model. It contains `process` subcomponents, which model partitions.

For each partition, we generate code to configure and deploy the system. We browse all partition's components and generate their code. For each kind of component, we apply a **code-generation pattern** describing the tokens created in the target language (functions, variables, etc.). These patterns are summarized in table 1.

Component	Code-generation pattern
System	Create directory with code from subcomponents
Processor	Configure runtime requirements of the kernel (scheduling, execution platform, etc.)
Process	Create a partition according to memory requirements
Virtual Processor	Configure runtime internals of the partition
Thread	Create stub functions and declare the thread in the partition.
Connection	Declare communication ports in the contained component (port type, timing requirements, etc.)
Data	Declare a new type and create mutexes according to their scheduling requirements in case of a protected data

Table 1. Code-generation patterns

For each system, we look for `process` and `processor`. The `processor` component configures scheduling and runtime requirements of the node (algorithm that schedules partitions, middleware functionnalities, etc.).

For each `process` component and its associated `virtual processor` we instanciate a new partition according to its requirements. We also create configuration code associated with the partition to set up its address space, the timeslice for the execution of its threads and all its requirements (scheduling algorithm for its tasks, etc.).

For each `thread` inside the partition, we instantiate a task and create its associated resources (mutexes, semaphores, intra-partition communication, etc.). If the thread communicates with other entities, we generate the appropriate function (also known as stub or skel) to send/receive data. These stubs or skels call functional code provided by the developer (shown as *user code* in figure 1).

For each data component, a new type is created. This type is mapped to a new one suitable for the underlying platform. In case of protected data, we automaticallu create mutexes and set up its appropriate scheduling requirements such as priority or ceiling protocol.

Finally, we inspect connections between components and generate their instantiation as well as their configuration. We also set up timing properties associated with them (deadline, refresh period, etc.).

3.3 Case study, metrics and results

So far, most of the code-generation patterns are implemented in our AADL toolsuite [10]. Some features (such as inter-partition communication) are still in development.

Two examples have been used to validate our approach as a proof of concept. The first one (*"basic"*) defines two partitions that periodically output a message. It shows that the generated code enforced model's requirements (for example, tasks' periods are correct). The second one (*"ping"*) defines a single partition with intra-partition communication (a threads sends a message to another thread located in the same partition). It demonstrates that communication resources are well allocated as well as messages sending and receiving. Other kinds of tests are still in development, such as the verification of partitions against failures and attacks.

Two kind of metrics could be used to demonstrate the correctness of our approach: the conformance of the generated code with the model and its compliance with embedded requirements. Some experiments demonstrate the correctness of our approach regarding usual requirements for embedded software.

Table 2 summarizes the size of code in term of lines of code and object files size. We used GCC and stripped binaries. For each example, we take metrics on each software component involved in our development process (figure 1): application code (provided by the developer), generated code, partition runtime and kernel.

Test	Component	SLOCs	.o size (Kbytes)
Basic	Application	12	0.097
Basic	Generated code	31	1
Basic	Partition runtime	2171	27.3
Basic	Kernel	4933	35
Total Basic		**7147**	**63.397**
Ping	Application	10	0.632
Ping	Generated code	180	2,5
Ping	Partition runtime	2171	29
Ping	Kernel	5629	35.7
Total Ping		**7990**	**67.832**

Table 2. Size of generated code and underlying executive for each test

Memory footprint is quite small, which is suitable for embedded hardware. The size of the kernel and partition runtime may vary, depending on included functions, but differences are not significant. On the contrary, the size of application and generated code depend on the AADL model's size. Tests presented here are small, but as soon as models will be bigger, size of generated code will grow significantly up. However, it will remain acceptable for embedded devices : in both examples, we set a single partition with two threads. Even with several partitions, the size of generated code would be acceptable and stay under few kilobytes.

4 Conclusion and future work

This article presents an approach to automatically generate code from AADL models targetting partitioned architectures. Both kernels and partition's code are automatically produced, configured and deployed.

Our code generator is based on transformation patterns that preserve security and safety requirements of the model. It improves reliability and system's robustness.

This work remains in a work-in-progress state and experimentation must be performed to improve our strategies and evaluate the impact of code generation on larger examples.

To increase the confidence on the generated code, we can use AADL specifications to generate testsuites. These could then be used as a first testbench for the final application.

We also plan to prove the correctness of our code generation patterns to ease the certification of generated applications. To do so, code generation patterns could be adapted to produce formal specifications for verification purpose.

Acknowledgement: This work has been funded in part by the ANR Flex-eWare project.

References

[1] ARINC. Draft 3 of Supp 1 to ARINC Specification 653: Avionics Application Software Standard Interface, 2003.

[2] J. Delange, J. Hugues, L. Pautet, and B. Zalila. Code Generation Strategies from AADL Architectural Descriptions Targeting the High Integrity Domain. In *4th European Congress ERTS*, Toulouse, 2008.

[3] P. Feiler, D. de Niz, C. Raistrick, and B. Lewis. From pims to psms. *Engineering Complex Computer Systems. 12th IEEE International Conference*, pages 365–370, July 2007.

[4] J. Hansson, P. H. Feiler, and J. Morley. Building secure systems using model-based engineering and architectural models. *Crosstalk*, pages 10–14, September 2008.

[5] S.-W. Lin, C.-H. Tseng, T.-Y. Lee, and J.-M. Fu. Vertaf: An application framework for the design and verification of embedded real-time software. *IEEE Trans. Softw. Eng.*, 30(10):656–674, 2004.

[6] J. Mason, K. Luecke, and J. Luke. Device Drivers in Time and Space Partitioned Operating Systems. *25th Digital Avionics Systems Conference*, pages 1–9, Oct. 2006.

[7] D. P. G. Peter H. Feiler and J. J. Hudak. The Architecture Analysis and Design Language (AADL) : An Introduction. Technical report, February 2006.

[8] A.-E. Rugina, K. Kanoun, and M. Kaaniche. An architecture-based dependability modeling framework using aadl. In *10th IASTED International Conference on Software Engineering and Applications (SEA'2006)*, 2006.

[9] J. Rushby. Partitioning for avionics architectures: Requirements, mechanisms, and assurance. NASA Contractor Report CR-1999-209347, NASA Langley Research Center, June 1999.

[10] B. Zalila, J. Hugues, and L. Pautet. *Ocarina user guide - http://aadl.enst.fr/ocarina*. TELECOM ParisTech, 2008.